Appears in *Proceedings, IASTED International Conference on Software Engineering and Applications (SEA) 2002*,
November 4-6, 2002, Cambridge, MA, USA.

REWRITING REQUIREMENTS FOR DESIGN

James Kirby, Jr.
Code 5546
Naval Research Laboratory
Washington, DC 20375 USA

ABSTRACT

Maintaining consistency between requirements and the design developed to satisfy them is both important and difficult. Maintaining consistency is important to satisfying stakeholders' desires, which the requirements express. Much of the difficulty of maintaining consistency stems from having redundant descriptions of requirements decisions—one in the requirements document and a second in the design document—typically recorded in widely divergent languages. To ameliorate this problem, we write requirements and design in such a way that requirements decisions and their expression in the requirements document are incorporated directly into the design document, which organizes the decisions and includes additional—design—decisions.

KEY WORDS

Software Requirements, Software Design, Software Methodologies, Software Engineering

1. Introduction

Often in development, decisions about required behavior are written down several times—e.g., in requirements, design, code—usually in different notations and languages (e.g., prose, UML, C). For stakeholders to have confidence that their needs—expressed in requirements—will be met, developers must be able to maintain and demonstrate consistency between requirements and the design. Maintaining and demonstrating the consistency of multiple representations of requirements decisions, in different languages poses a significant challenge [1].

This paper describes an approach to ensuring consistency between requirements and design that involves representing the decisions made in developing requirements and design in a way that should reduce the effort required to demonstrate and maintain consistency. The next section discusses A-7 software requirements and design and the refinements that several teams of researchers have made to the requirements model. Subsequent sections discuss the light control example; the system requirements, system design, and software design of the light control system; and why we have chosen this particular organization.

2. A-7 and SCR

The approach described in this paper builds on the A-7 and SCR work in requirements and design. This section begins with a discussion of A-7 software requirements and design. While the term *SCR* (for Software Cost Reduction) was coined to refer to this work, we reserve the term *SCR* for the subsequent research into requirements and system design that is described following the discussion of the A-7 work (though some of those researchers don't use the term themselves).

2.1. A-7 Software Requirements

While there is more to A-7 requirements [2, 3], this paper focuses on its recording of behavioral requirements, the externally visible behavior the software is required to exhibit. The sections of the A-7 requirements that contribute to recording behavioral requirements are Hardware Interfaces, Modes of Operation, Software Functions, and Glossary. For each device input or output that changes value independently of other inputs or outputs, Hardware Interfaces contains an input or output variable declaration. The values the software sends to the output variables is the required behavior of the software.

Software Functions describes how the software should set the output variables—the required behavior of the software. For each output there is one software function that specifies its value. To avoid describing an implementation, the software functions are written in terms of conditions and events defined on aircraft operating conditions. "*Conditions* are predicates that characterize some aspect of the system for a measurable period of time." "An *event* occurs when the value of a condition changes from true to false or vice versa." [3] For example, *altitude > 500* is a condition that is true while aircraft altitude is greater than 500 feet and is false otherwise. The event *@T(altitude > 500)* denotes the instant when the condition *altitude > 500* becomes true.

To capture the software functions' dependence on state, Modes of Operations defines classes of system state, called *modes*. The history of execution determines the mode of the system. Modes of Operation gives "the initial mode of the system and the set of events that cause transition between any pair of modes" [3].

Special tables specify the functions in Software Functions. They are written in such a way that it is easy to inspect them for certain types of common errors, e.g., incompleteness, inconsistency. When the system mode and the truth of one of several conditions determines the value of an output variable, a condition table specifies the func-

tion.When the system mode and occurrences of particular events determine the value of an output variable, an event table specifies the function.

2.2. A-7 Software Design

This paper focuses on a portion of the A-7 design: on the decomposition of the software into components—called *modules*—and on the behavior of each component, which are described, respectively, by a software module guide and a set of specifications of module behavior. The information hiding principle [4] guides the decomposition described in the software module guide. Each module is characterized by a *secret*, a decision that designers judge is likely to change independently. It is the responsibility of the module to encapsulate or hide the decision. When the decision changes, only the module that hides the decision should be affected (or a small number of easily identifiable modules).

The structure of the module guide can be represented as a tree that decomposes the software into modules. Each module is a work assignment for one or more programmers. The software is decomposed until each leaf module constitutes a suitable work assignment. A non-leaf module is composed of the set of work assignments described by the modules that are its children in the tree.

The secret of each module—the decision that it hides from other modules—distinguishes its responsibility from that of other modules. Listed below are the secrets of the top three modules for the A-7 [5].

- **Hardware-Hiding Module.** The Hardware-Hiding Module includes the programs that need to be changed if any part of the hardware is replaced by a new unit with a different hardware/software interface but with the same general capabilities. This module implements "virtual hardware" or an abstract device that is used by the rest of the software. The primary secrets of this module are the hardware/software interfaces. The secondary secrets of this module are the data structures and algorithms used to implement the virtual hardware.
- **Behavior-Hiding Module.** The Behavior-Hiding Module includes programs that need to be changed if there are changes in the sections of the requirements document that describe the required behavior. The content of those sections is the primary secret of this module. These programs determine the values to be sent to the virtual output devices provided by the Hardware-Hiding Module.
- **Software Decision Module.** The Software Decision Module hides software design decisions that are based upon mathematical theorems, physical facts, and programming considerations such as algorithmic efficiency and accuracy. The secrets of this module are **not** described in the requirements document.

Together, the three modules constitute the complete set of work assignments for the A-7 programmers. Note that they take responsibility for, respectively, the A-7 hardware, the externally visible behavior of the A-7 software, and everything else (i.e., decisions left to the software designers). The remainder of this paper shall not be concerned with the Software Decision Module (the "everything else").

Each specification of module behavior specifies the behavior of a set of programs. The set of programs may either consist of programs that are usable by other modules, in which case the programs are called the *interface* of the module and the specification is referred to as an *abstract interface*, or they may consist of programs that use programs in other modules to set the values of the virtual device output variables (required externally visible behavior of the software), in which case the specification is referred to as a *function driver.*

This paper focuses on modules responsible for hiding decisions recorded by the requirements: Device Interface (which is a submodule of Hardware-Hiding), System Value, Mode Determination, and Function Driver (which are submodules of Behavior-Hiding). Each submodule of the Device Interface Module defines a virtual device that hides details of a physical device described in the requirements that the designers judged were likely to change independently. There need not be a one-to-one correspondence between a physical device and a virtual device. One physical device that has two aspects the designers judge are likely to change independently may be represented by two virtual devices. Two physically distinct devices that are always replaced together may be represented by a single virtual device. The programs on the abstract interface allow users to accept inputs from and to send outputs to a device. The interface can also notify users of specified changes in the device (e.g., device failure) or in a value that it measures (e.g., for an altimeter, altitude becomes greater than 500 feet). Each submodule defines a set of terms that describe aircraft operating conditions and the state of the aircraft's devices. The abstract interface uses the terms to describe the inputs from devices and the effects of outputs sent to devices. These terms are also used by specifications of other modules (e.g., Function Driver submodules).

Submodules of the System Value Module compute values that can be used by other modules. The submodules hide the rules in the requirements that define the values. Like the Device Interface Module, all the values provided by the module represent either aircraft operating conditions (e.g, the latitude and longitude of some point on the earth, how long the aircraft will take to fly to some specified point on the earth) or the state of some device (e.g., whether "the Air Data computer is functioning and producing current and reasonable altitude readings" [6]). And like Device Interface, the submodule defines terms representing aircraft operating conditions and device state that it uses to describe the values it returns to its users.

The interface of the Mode Determination Module provides programs that allow users to determine whether the

system is in a particular mode and that notify users when the system enters or exits a specified mode. The primary secrets of the module are the rules describing mode transitions in the requirements.

The Function Driver specifications, which hide the rules recorded in requirements determining the values of the output variables, differ from other module specifications in that they do not have an abstract interface: function drivers are not used by other programs. While their form is similar to the table functions in the A-7 software requirements, they differ from the software requirement functions in several ways. First, a function driver controls virtual outputs (as opposed to the physical—though abstract—outputs identified in the software requirements specification) for a virtual device defined by a submodule of Device Interface. Second, a function driver explicitly identifies the Device Interface program that sets the virtual device output variable whose value it specifies. Third, instead of referring to the terms describing aircraft operating conditions that the software requirements' Glossary defines, function drivers refer to analogous terms that other module specifications define.

2.3. SCR System Requirements and System Design

In the Four-Variable Model described in [7], Parnas and Madey abstract from the A-7 software requirements model and extend it to cover system requirements. Mathematical variables representing physical quantities in the system environment replace device inputs and outputs for describing required behavior. Mathematical relations on vectors of time functions replace the conditions, events, modes, and tables of the A-7 requirements. Parnas and Madey define a System Requirements Document that specifies the externally visible behavior required of the system, and a System Design Document that specifies the behavior required of the system's peripheral devices (Parnas and Madey consider systems with several computers and describe them and communication among them in the System Design Document; this paper assumes a single computer).

Writing in terms of physical quantities in the system's environment describes system—as opposed to software—behavior. Instead of specifying the values the software should send to the output devices, the model specifies the required values of mathematical variables, called *controlled variables*, that denote physical quantities in the system's environment (e.g., the position of an actuator, the location of an icon in a display to an operator) that the system must control. Parnas and Madey specify the required values of the controlled variables in terms of mathematical variables, called *monitored variables*, which also denote physical quantities in the system's environment. They capture system design decisions by specifying the behavior of the input and output devices. Similar to the A-7 model, the specification defines input and output variables. Specifying the values of the inputs in terms of the monitored variables, and the effects of the outputs in terms of the controlled variables complete the specification of device behavior.

Instead of using tabular functions to specify required behavior, [7] leaves open the form that the functions describing required behavior of particular systems should take. To describe precisely the information that the System Requirements and System Design documents record, Parnas and Madey define four relations among the monitored, controlled, input, and output variables. The relation

$$\text{REQ: } M \rightarrow C,$$

a relation from all possible histories (where *possible* means allowed by environmental constraints) of the monitored variables to all possible histories of the controlled variables, describes required system behavior. *M*, the domain of *REQ*, is a set of vectors. For each monitored variable, a vector has one element, a time function. The time function, which specifies the value of the monitored variable as a function of time, describes a possible history of that monitored variable. Each vector of monitored variable time functions describes a possible history of all of the monitored variables. *M* is the set of all possible histories of the monitored variables. *C*, the range of *REQ*, is a similar set of vectors of time functions specifying possible histories of the controlled variables. For each possible history of the monitored variables in the set *M*, *REQ* specifies one or more possible histories of the controlled variables in the set *C*. *REQ* is a relation and not a function to allow for the small errors inherent to real implementations. Below, this paper will use similar relations on vectors of time functions for other variables to describe the contents of other documents.

It is often useful to specify constraints that the environment of the system (e.g, physical laws, the behavior of other systems) imposes on its behavior. The relation

$$\text{NAT: } M \rightarrow C$$

captures those constraints. For each possible history of the monitored variables, it specifies which possible histories of the controlled variables the environment allows.

Similar to *M* and *C*, *I* and *O* are sets of possible histories of the system's input and output devices, respectively. The relation

$$\text{IN: } M \rightarrow I$$

specifies the behavior of the input devices; the relation

$$\text{OUT: } O \rightarrow C$$

specifies the behavior of the output devices.

To provide a more precise semantics of SCR specifications and to facilitate mechanized analysis of specifications, Heitmeyer and colleagues developed a finite state model for SCR [8, 9] and have developed tools for creating and analyzing SCR specifications. The work assumes the basic notation and format for specifying required behavior intro-

duced by the A-7 (e.g., conditions, events, terms, modes, tables) and applies it in the context of the Four-Variable model (e.g., specifying required system behavior in terms of monitored and controlled variables).

3. The Light Control System

The example in this paper is based upon the Light Control System (LCS) case study [10] reported by Heitmeyer and Bharadwaj in [11]. The LCS controls the lights in the rooms and corridors of a building, attempting to keep energy use low by making use of natural light when possible and by turning off lights in unused rooms. As in [11], this paper is concerned with specifying light control for a single, typical office. Each office has a window, wall lights, and window lights. Users can determine office brightness when the room is occupied and when it is temporarily unoccupied. The system uses door and motion sensors to determine when a room is empty, and an outdoor light sensor to determine how much light is provided by the office window. A panel in the facility manager's office alerts him to the failure of some sensors, allows him to override some settings, and to determine how long an office should be empty to be considered unoccupied.

The following two sections discuss our system requirements, system design, and software design specifications of the LCS.

4. System Requirements and System Design of the LCS

The LCS System Requirements Document is a typical small SCR requirements document. Six functions specify the required values of six controlled variables denoting whether wall and window lights are on, their brightness, and the color of two lights indicating sensor malfunctions. Table 1 illustrates several controlled variable declarations, listing the name and type of each variable and describing how to interpret the values. The functions are written in terms of 14 monitored variables (see Table 2), one mode class, and five terms. The monitored variables include variables denoting light settings chosen by the office occupant, default settings, ambient light level in the office, whether the office is occupied, whether values of certain variables are unobtainable, and time. The terms denote current light settings in the office, whether the facilities manager has overridden the office settings, and how much light must be provided by the office lights. Table 3 provides an example

Table 1: Selected Controlled Variables

Variable	Type	Physical Interpretation
cWallLL	yLightLevel	cWallLL denotes the brightness of the office wall lights.
cWallLights	yLight	cWallLights = on iff the office wall lights are lit.

Table 2: Selected Monitored Variables

Variable	Type	Physical Interpretation
mChosenLSVal	yLightLevel	mChosenLSVal denotes the value indicated by the slider labelled "Chosen Light Scene Value" in the office control panel.
mWallLights	yLight	mWallLights = on iff someone is pressing the button labelled "Wall Lights" in the office control panel.

Table 3: Required Value of cWallights

	Events	
	@T(mWallLights = on) when (cWallLights = off) or @T(mcStatus = occupied)	@T(mWallLights = off) when cWallLights = on) or @T(mcStatus = unoccupied) or @T(mFMOverride) when (not mcStatus = occupied)
cWallLights =	on	off

specification of the required value of cWallLights, whose declaration appears in Table 1. The first column of the event table describes when cWallLights assumes the value *on*; the second describes when it assumes the value *off*.

The LCS System Design Document diverges from the model presented in [7] which uses abstract inputs and outputs and monitored and controlled variables to specify the behavior of *physical* devices. In contrast, the LCS system design specifies the behavior of *virtual* devices which abstract from details of the physical devices that the developers judge are likely to change. The system design uses virtual device inputs, virtual device outputs, and monitored and controlled variables to specify the virtual device behavior. We replace the *IN* relation of Parnas and Madey with IN_v, the virtual *IN* relation which specifies the behavior of the virtual device input variables, I_v

$$IN_v: M \rightarrow I_v$$

and we replace the *OUT* relation with OUT_v, the virtual *OUT* relation which specifies the behavior of the virtual device output variables, O_v

$$OUT_v: O_v \rightarrow C$$

The LCS System Design Document consists of dictionaries declaring the types, monitored and controlled variables, and virtual device input and output variables (Table 4 illustrates the declaration of a virtual output variable). Functions from virtual device output variables to controlled variables specify the effects of setting the virtual outputs. Functions from monitored variables to virtual device input

variables specify the value of the virtual inputs. Table 5 is a

Table 4: Virtual Output Declaration

Name	Type
ovWallLight	Boolean

Table 5: Effect of ovWallLights

	Conditions	
	ovWallLights	NOT ovWallLights
cWallLights =	on	off

condition table specifying the value of *cWallLights* as a function of the virtual output *ovWallLights*. When the Boolean *ovWallLights* is *true*, *cWallLights* is *on*; otherwise it is *off*.

5. Software Design of the LCS

Our software design of the LCS, modeled on the A-7 design, consists of a software module guide and a set of specifications of module behavior. Table 6 outlines the module structure in the LCS software module guide. Only modules hiding decisions recorded in the System Requirements and System Design documents are included in our LCS design. Modules whose names are listed in **bold** were lifted whole from the A-7 module guide. Their secrets, and hence their responsibilities, were minimally adapted to the LCS (with exceptions discussed below). Modules whose names are in *italics* represent work to be done; the list includes them to show what has been left out of this paper. The remaining—leaf—modules, indicated by regular type, provide a set of virtual devices (see submodules of the Device Interface Module), specify the required values of malfunction lights and office lights (see submodules of Function Drivers), provide a means to determine the current system mode (Mode Determination Module), and provide a means to determine the value of monitored variables and terms and to set the value of controlled variables (see System Value Module).

System Value and Function Driver are the modules whose secrets are most affected by changes in SCR requirements as compared to A-7 requirements. Both reflect the SCR specification of required values of controlled variables, which represent physical quantities, as opposed to the specification of the values of output devices in A-7 requirements. The change to the Function Driver secret is a wording change from referring to *outputs* to referring to *controlled variables*. The change to the System Value secret is more substantial: the module must not only provide values to its users, it must also allow its users to set the values of controlled variables. This was not needed in the A-7 design since the Function Driver submodules set the output devices directly. In the revised SCR design, Function Driver submodules set the value of controlled variables. System Value hides how to accomplish the changes to the controlled variable values.

Table 6: LCS Module Decomposition

1. Hardware Hiding
- *1.1. Extended Computer*
- **1.2. Device Interface Module**
  - 1.2.1. Door Closed Contact
  - 1.2.2. Facilities Manager Console
  - 1.2.3. Motion Detector
  - 1.2.4. Office Control Panel
  - 1.2.5. Outdoor Light Sensor
  - 1.2.6. Timer
  - 1.2.7. Wall Lights
  - 1.2.8. Window Lights

2. Behavior Hiding
- **2.1. Function Drivers**
  - 2.1.1. Malfunction Lights
  - 2.1.2. Office Lights
- **2.2. Shared Services**
  - 2.2.1. Mode Determination
  - 2.2.2. System Value

3. Software Decision

Specifications of submodules of the Device Interface Module, which hide from their users how the physical devices are likely to change, all have a similar form (see Table 7). Each is an abstract interface defining a virtual

Table 7: Device Interface Abstract Interface

Section	Contents
Output Variables	Declaration of I_v, virtual device input variables
Value of Output Variables	IN_V: $M \rightarrow I_V$
Input Variables	Declaration of O_v, virtual device output variables
Effects of Input Variables	OUT_V: $O_V \rightarrow C$
Access Programs and Events Signalled	Protocol for using the module
Dictionaries	Types, monitored variables, controlled variables

device. Each abstract interface lists the declarations of the virtual device input variables (denoted I_v in the Contents column to distinguish them from the physical inputs that I denotes) which are outputs to users of the interface. Functions (which comprise IN_v) specify the values of the virtual inputs in terms of monitored variables. The interface also lists the declarations of the virtual device output variables (O_v) which are inputs from users (see Table 4). Functions specifying the values of controlled variables in terms of the virtual device output variables (which comprise OUT_v) describe the effects of users setting the virtual output variables (see Table 5). An access program table defines the

protocol that user programs follow to obtain the value of virtual device inputs and to set the value of virtual device outputs. Table 8 specifies a program which accepts as an input a value whose type is described by the declaration of *ovWallLights* (see Table 4); Table 5 specifies the effect of calling *sovWallLights*. Standard SCR event notation defines the events signalled by the abstract interface in terms of virtual device input variables. The complete set of Device Interface abstract interfaces is followed by dictionaries defining types, monitored variables, and controlled variables referenced in the abstract interfaces.

Table 8: Device Interface Program Table Entry

Program	Parameters	Description
sovWallLights	input	ovWallLights

The organization of the System Value Module abstract interface (see Table 9), which hides how to determine the value of monitored variables and certain terms and how to set the value of controlled variables, is similar to that of Device Interface. The interface lists declarations of monitored variables and terms, which are outputs to users. In contrast to the outputs of the Device Interface abstract interface, no functions specify the value of the outputs; the physical interpretation provided by the declarations describes the values of the monitored variables (see Table 2) and terms. The abstract interface lists the declarations of controlled variables which are the inputs from users. As with the monitored variables and terms, no functions specify the effects of setting the controlled variables; the physical interpretations provided by the controlled variable declarations describe the effect of users setting the values of the controlled variables (see Table 1). An access program table defines the protocol that user programs follow to obtain the value of monitored variables and terms and to set the value of controlled variables. Table 10 provides an example specification of the program *sWallLights*, which accepts an input of type *yLight* (*on* or *off*) as specified by Table 1. The effect of the program is to set the wall lights in the office on or off, as described by the physical interpretation of *cWallLights*. Standard SCR event notation defines the events signalled by the abstract interface in terms of controlled variables and terms. The abstract interface concludes with a dictionary defining the types used.

Table 9: System Value Abstract Interface

Section	Contents
Output Variables	Declarations of monitored variables (*M*) and terms
Input Variables	Declarations of *C*, controlled variables
Access Programs and Events Signalled	Protocol for using the module
Dictionaries	Types

Table 10: System Value Program Table Entry

Program	Parameters	Description
sWallLights	input	cWallLights

Each function driver specification (see Table 11), which hides the rules that determine the required value of the controlled variables, lists the declarations of the controlled variables set by the function driver and the functions that specify the values of the controlled variables (see Table 3). The function driver specification includes dictionaries declaring the types, modes, monitored variables, and terms used by the function.

Table 11: Function Driver Specification

Section	Contents
Controlled Variables	Declarations of *C*, controlled variables
Behavior	REQ: M $\rightarrow$ C
Dictionaries	Types, modes, monitored variables (*C*), terms

Table 12 lists the contents of the mode determination module which hide the rules determining the current system modes (there may be multiple mode classes, allowing the system to be in several modes simultaneously). The current mode of each declared mode class are the outputs to users. An access program table defines the protocol that user programs follow to obtain the current mode of each mode class. SCR event notation defines the events signalled by the abstract interface in terms of the mode classes.

Table 12: Mode Determination Abstract Interface

Section	Contents
Output Variables	Declarations of mode classes
Access Programs and Events Signalled	Protocol for using the module

6. Discussion

The approach to documenting requirements and design discussed above partitions the decisions of development in such a way that distinguishing among system requirements, system design, and software design decisions, and recording and organizing those decisions is a relatively straight forward task. Separation of concerns serves to decompose the complex and difficult activity of system development into a set of relatively distinct and simpler activities. The nature of the decomposition eliminates some of the redundancy by using the same descriptions of behavior in requirements and design by organizing development docu-

mentation in such a way that where there is overlap of requirements and design decisions, those decisions are recorded once and shared among the work products. This should eliminate much effort required to keep consistent redundant representations of requirements decisions recorded in widely divergent languages.

We have adapted the specifications of modules that hide system requirements and design decisions—Function Driver, Device Interface, Mode Determination, System Value—so that the declarations of monitored and controlled variables, terms, virtual inputs and outputs, and the table functions that specify their values in the modules are identical in contents and notation to how the same decisions are recorded in the system requirements and system design. Thus engineers do not need to document system requirements and design decisions again in the software design. Nor do they need another notation for capturing those decisions. For the documents that we discuss, the concept of identity replaces the concept of traceability.

We have diverged from the Four-Variable Model of [7] by specifying the system design in terms of virtual devices, which hide aspects of the physical devices that are likely to change, rather than in terms of physical devices. This can be useful since development projects often choose devices fairly late, when software designs have been developed based on designers' best guesses. And even when devices are chosen early, the choices can be changed for many reasons, including the inability of the vendor to deliver, or the unexpected availability of a cheaper or more effective device. Specifying system design in terms of virtual devices should make it easier to adapt the system and software design documents and the software to such changes. Basing system design on virtual devices allows system and software design to progress while reducing the risk of significant impact making or changing selections of physical devices.

We have diverged from the A-7 model of software design in several ways. We have adapted the software module guide by identifying a module that hides how to set the value of controlled variables. We have adapted the specifications of modules that hide system requirements and design decisions—function drivers, device interface modules, and System Value—so that the declarations of monitored and controlled variables, terms, virtual inputs and outputs, and the table functions that specify their values in the modules are identical in contents and notation to how the same decisions are recorded in the system requirements and system design.

Acknowledgments

The seeds of this paper were comments made over a decade ago by Dave Weiss and a more recent paper by Connie Heitmeyer and Ramesh Bharadwaj [11]. Thoughtful and detailed suggestions from Heitmeyer, Bharadwaj, Weiss, and anonymous reviewers led to improvements in the paper. The LCS System Requirement, System Design, and Software Design Documents described in this paper were derived from the LCS specifications described in [11].

REFERENCES

[1] A. Kozlenkov, A. Zisman, Are their Design Specifications Consistent with our Requirements?, *IEEE Joint International Requirements Engineering Conference*,2002, 145-154.

[2] T.A. Alspaugh, S.R. Faulk, K.H. Britton, R.A. Parker, D.L. Parnas, J.E. Shore, *Software Requirements for the A-7E Aircraft*, NRL Final Report 5530, 1992.

[3] K.L. Heninger, Specifying Software Requirements for Complex Systems: New Techniques and Their Application, *IEEE Trans. on Softw. Eng.*, SE-6(1), 1980, 2-12.

[4] D.L., Parnas, On the Criteria to be Used in Decomposing Systems in Modules, *Comm. of the ACM*, 15(12), 1972, 1053-1058.

[5] K.H. Britton, D.L. Parnas, *A-7E Software Module Guide*, NRL Memorandum Report 4702, 1981.

[6] P.C. Clements, *Abstract Interface Specifications for the A-7E Shared Services Module,* NRL Memorandum Report 4863, 1982.

[7] D.L. Parnas, J. Madey, Function Documents for Computer Systems, *Science of Computer Programming*, 25(1), 2995, 41-61.

[8] C.L. Heitmeyer, Software Cost Reduction, in J.J. Marciniak (Ed.), *Encyclopedia of Software Engineering*, (New York: John Wiley, 2002), 1374-1380.

[9] C.L. Heitmeyer, R.D. Jeffords, B. Labaw, Automated Consistency Checking of Requirements Specifications, *ACM Trans. on Softw. Eng. and Meth.*, 5(3), 1996, 231-261.

[10] The Light Control Case Study: Problem Description, *Journal of Universal Computer Science*, 6(7), 2000.

[11] C. Heitmeyer, R. Bharadwaj, Applying the SCR Requirements Method to the Light Control Case Study, *Journal of Universal Computer Science*, 6(7), 2000, 650-678.